**5-8整数变换问题**

问题描述：关于整数i的变换f和g定义如下：f(i)=3\*i;g(i)=[i/2];

试设计一个算法，对于给定的两个整数n和m，用最少的f和g变换次数将n转化为m。

例如：可以将整数15用四次变换将它变成整数4:4=gfgg(15)。当整数n不可能变换成m时怎么处理？

算法设计：对任意给定的整数n和m，计算将整数n转换成m所需要的最少变换次数。

数据输入：由文件input.txt给出输入数据。第一行有2个正整数n和m。

结果输出：将计算的最少变换次数以及相应的变换序列输出到文件output.txt。文件的第一行是最少变换次数，第2行是相应的变换序列。

输入文件： 输出文件

input.txt output.txt

15 4 4

gfgg

二、问题分析

A.题目分析：

观察f和g两个函数发现，f总是使得i变大，g总是使得i变小。因此我们在决定让x执行哪个函数之前必须先判断i和目标值m之间的大小关系。如果x>m，就让其执行g函数；反之，执行f函数。

这道题目有两种情况，一种是有解，即n可以通过函数变换成m；另一种是无解，即n无法通过函数变换成m。那我们如何去判断某一个样例输入是属于哪种情况呢？

第一种情况比较容易，即我们只需要判断最后的i是否等于m即可。如果i等于m，那么说明n已经被变换成m了，递归返回。

第二种情况的话需要我们找下规律。假设我们的输入n=9,m=5。

n>m,执行g,n=[9/2]=4

n<m,执行f,n=3\*4=12

n>m,执行g,n=[12/2]=6

n>m,执行f,n=[6/2]=3

n<m,执行g,n=3\*3=9

n>m,执行f,n=[9/2]=4

我们会发现如果n的值陷入了一个重复的循环。也就是如果在我们递归的过程，出现了我们前面计算过的元素，那就说明n是无法转换成m的。

B.算法设计：

用回溯法解决整数变换问题，显然用子集树是最合适的。该子集树如下

我们并不知道该树有多少层，因为对于不同整数，它们之间所需要的变换次数也是不同并且不确定的。那么我们这个时候不能像以往的回溯法，用进行的层数来作为返回条件。

根据我们前面的题目分析，可以知道我们返回条件有两个，一个是i等于了m，也就是if(i==m);另一个是出现了重复的数字。这个返回条件我们需要构造一个test方法来判断。这个test方法主要就是在下一层走之前我们先对这个参数和以往的参数进行一个比较，如果这个参数和以往的某个参数相等，说明它已经出现过了，那么我们就让回溯递归得函数返回0，说明n是无法转换为m的。

剪枝条件：

显示约束：如果x>m，我们就剪掉它的左子树；如果x<m，我们就剪掉它的右子树；

隐式约束：如果我们在某次计算的过程中发现当前的计算次数已经大于或等于最少计算次数了，那么我们就剪掉这个分支。

打印最优解的路径（函数序列）：

在算出最少变换次数后我们还要打印出该变换路径所对应的函数序列。

这里，我们考虑用一个两个数组来记录。sign[]数组记录当前的路径，而minsign[]记录最少变换次数所对应的路径。sign[]和minsign[]都初始化为-1。如果值为0表示左子树f，如果值为1表示右子树g。这样子我们就可以用一个while循环遍历minsign，打印出相应的函数序列了。

三、程序代码：

#include<iostream>

#include<fstream>

#include<sstream>

using namespace std;

class IntChange{

public:

int n;//待变换的整数

int m;//需要变换成的整数

int nowcount;//当前已变换的次数

int mincount;//记录最小的变换次数

int \*sign;//记录路径，0表示左子树f,1表示右子树g

int \*minsign;//记录最小变换次数的路径

int \*recordx;//记录出现过的值

int Backtrack(int x);

int test(int x){

int i=1;

while(recordx[i]!=-1){

if(x==recordx[i]) return 0;

i++;

}

return 1;

}

int f(int i){

return 3\*i;

}

int g(int i){

return i/2;

}

};

int IntChange::Backtrack(int x)

{

if(test(x)==0){

return 0;

}//检测n是否能够转化为m

if(x==m){

if(nowcount<mincount){

mincount=nowcount;

for(int i=1;i<=n;i++) minsign[i]=sign[i];

}

return 1;//返回

}

nowcount++;

recordx[nowcount]=x;

if(nowcount<mincount){//当前的变换次数小于mincount

if(x<m){

sign[nowcount]=0;

Backtrack(f(x));//当x小于m时先进入左子树

}

else{

sign[nowcount]=1;

Backtrack(g(x));//当X大于m时进入右子树

}

}

}

void Compute(int n,int m,ofstream &outfile)

{

IntChange ic;//实例化一个对象

ic.n=n;

ic.m=m;

ic.nowcount=0;

ic.mincount=INT\_MAX;

ic.sign=new int[256];

ic.minsign=new int[256];

ic.recordx=new int[256];

for(int i=0;i<=n;i++){

ic.sign[i]=-1;ic.minsign[i]=-1;ic.recordx[i]=-1;//初始化为-1

}

if(ic.Backtrack(ic.n)==1){

outfile<<ic.mincount<<endl;

int i=1;

while(ic.minsign[i]!=-1){

if(ic.minsign[i]==0) outfile<<"f";

else outfile<<"g";

i++;

}

outfile<<endl;

}

else{

outfile<<ic.n<<"无法转化为"<<ic.m<<endl;

}

delete []ic.sign;

delete []ic.minsign;

}

int main(){

ifstream cinfile;

cinfile.open("input.txt",ios::in);

int n,m;

cinfile>>n>>m;

cinfile.close();

ofstream outfile;

outfile.open("output.txt",ios::out);

Compute(n,m,outfile);

outfile.close();

return 0;

}

四、分析与总结

(1).对于不知道层数的子集树，最重要的是确定它的返回条件，否则很容易出现死循环的问题。

(2).在做题时，用类去封装会使得代码更加简洁易懂。

(3).mincount应该初始化为INT\_MAX，而不能是0，否则会一直为0。